Stacks, Queues & Trees

Harald Gall, Prof. Dr.

Institut für Informatik

Universität Zürich

http://seal.ifi.uzh.ch

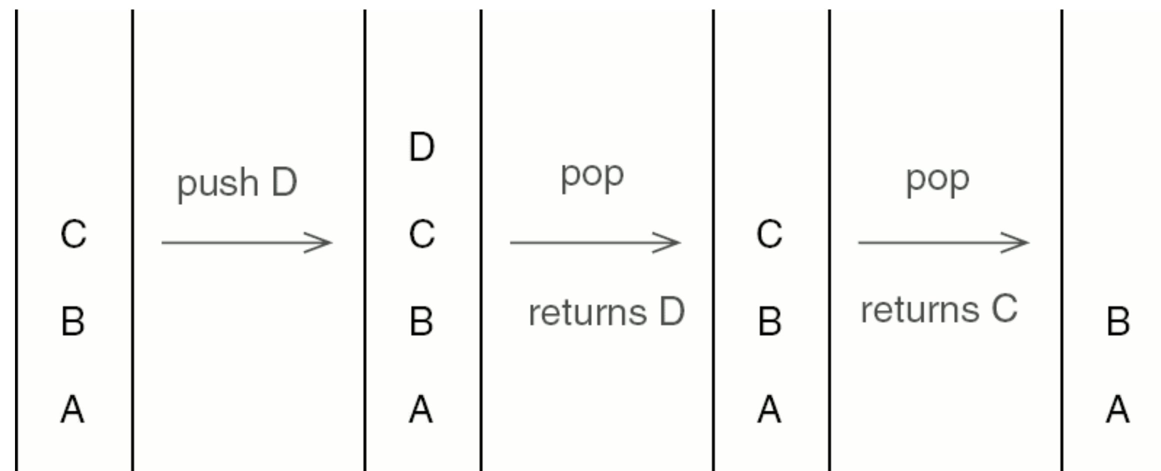
University of Zurich
Department of Informatics

The Stack Interface

- *Stack*
  - Data structure holding several items.
  - New items added to the top of the stack.
- *Pushing*
  - Push items onto the stack
- *Pop*
  - Pop items off of the stack or extract the topmost item.
- *Peek*
  - Peek at the top item or check if the stack is empty.

The Stack Interface

- *last-in, first-out* or *LIFO policy*
  - Last item pushed on the stack is next item popped off the stack



The Stack Interface

```
1 /** A last-in, first-out stack of Objects. */
2 public interface Stack {
3
4   /** Return true if this Stack is empty. */
5   public boolean isEmpty();
6
7   /**
8    * Return the top Object on this Stack. Do not modify the Stack.
9    * @throws EmptyStructureException if this Stack is empty.
10   */
11  public Object peek();
12
```

The Stack Interface

```
13   /**
14    * Remove and return the top Object on this Stack.
15    * @throws EmptyStructureException if this Stack is empty.
16    */
17   public Object pop();
18
19   /** Add target to the top of the Stack. */
20   public void push(Object target);
21
22 }
```

The Stack Interface

- Generics
  - The Stack interface describes a stack of Objects.
    - s.push(new Die(3));
  - When we pop a Stack we get back an Object.
  - We usually cast it to a more specific class so we can use specific functionality.
    - ((Die)(s.pop())).roll();
  - This is not only inconvenient, it is slightly unsafe.
  - We must remember what we put on the Stack, or we might try to cast something that is not really a Die to be a Die.

The Stack Interface

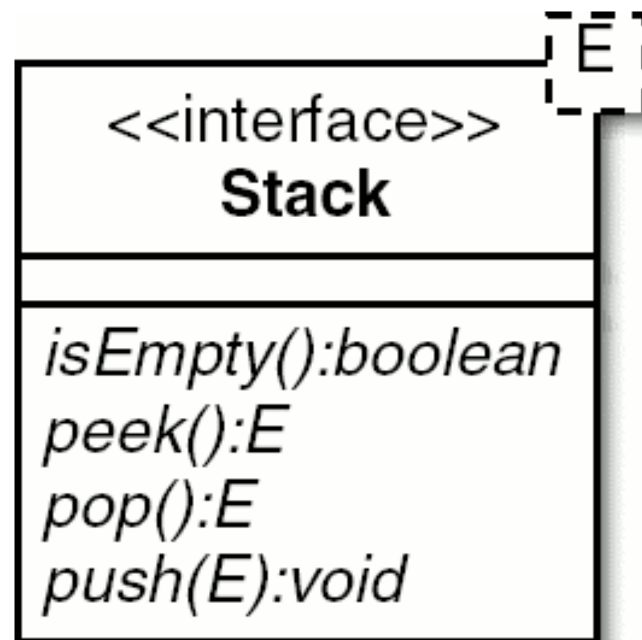
- *Generic* classes and interfaces
  - Has one or more ***type parameters***.
  - We have to specify a type for each type of parameter.
    - Stack<Die> s = new Stack<Die>():
    - s.push(new Die());
      s.pop().roll();
  - Java won't let us push anything of the wrong type onto this Stack.
  - The pop() method's return type is specified by the type parameter.

The Stack Interface

```
1 /** A last-in, first-out stack. */
2 public interface Stack<E> {
3
4   /** Return true if this Stack is empty. */
5   public boolean isEmpty();
6
7   /**
8    * Return the top item on this Stack, but do not modify the Stack.
9    * @throws EmptyStructureException if this Stack is empty.
10   */
11  public E peek();
12
13  /**
14   * Remove and return the top item on this Stack.
15   * @throws EmptyStructureException if this Stack is empty.
16   */
17  public E pop();
18
19  /** Add target to the top of the Stack. */
20  public void push(E target);
21
22 }
```

The Stack Interface

- In UML diagrams type parameters are shown as dashed boxes are at the upper right of a class.



The Call Stack

```
 1 /** Compute the hypotenuse of a right triangle. */
 2 public class Hypotenuse {
 3
 4   /** Return the square of the number x. */
 5   public static double square(double x) {
 6     return x * x;
 7   }
 8
 9   /**
10    * Return the hypotenuse of a right triangle with side lengths x
11    * and y.
12    */
13   public static double hypotenuse(double x, double y) {
```

The Call Stack

```
14        double x2 = square(x);
15        double y2 = square(y);
16        return Math.sqrt(x2 + y2);
17    }
18
19    /** Test the methods. */
20    public static void main(String[] args) {
21        double result = hypotenuse(3, 4);
22        System.out.println(result);
23    }
24
25 }
```

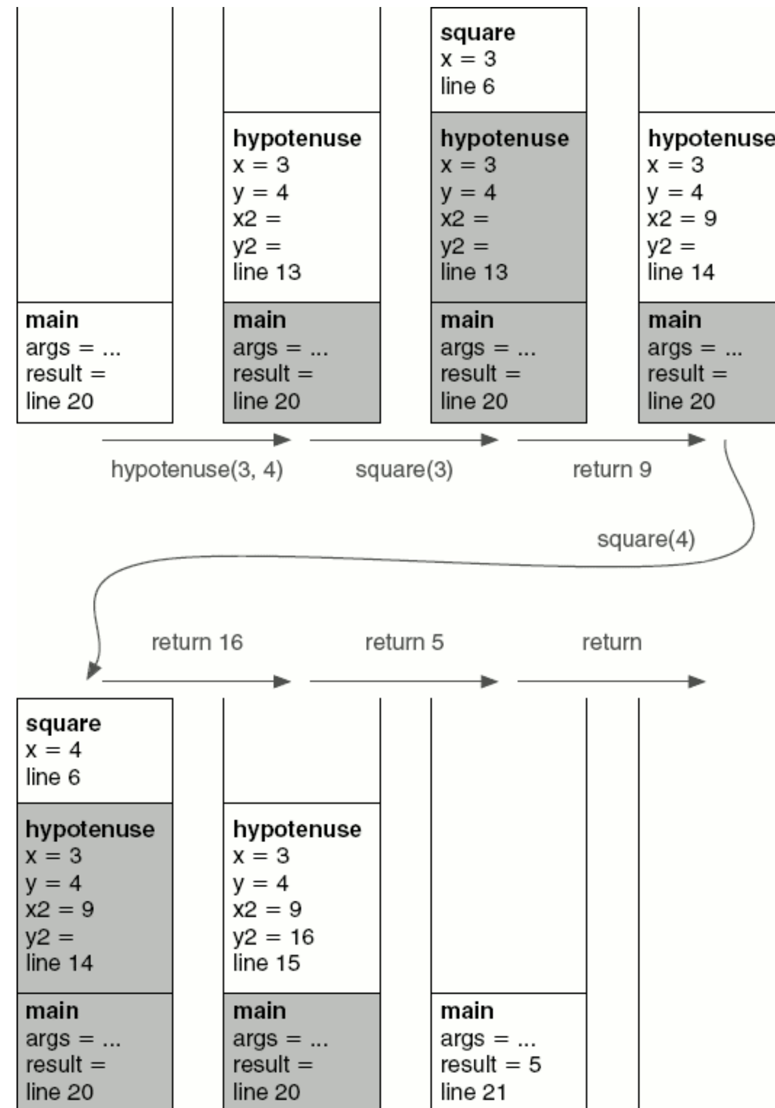
The Call Stack

- The hypotenuse() method invokes the square() method twice.
- How does Java know where to go when it finishes an invocation of square()?
- Java keeps track of what it was doing before the invocation started by using a stack.
  - This stack is called a ***call stack***.

The Call Stack

- Every time a method is invoked, a *call frame* is created.
  - The call frame keeps track of the current state of the method.
  - It stores any arguments or variables for the method.
  - It also keeps track of how far along the method has proceeded.

The Call Stack



The Call Stack

- Knowledge of the call stack helps us understand some of Java's error messages.
- The error message shows a *stack trace* (a snapshot of the call stack) indicating what was going on when the program crashed.

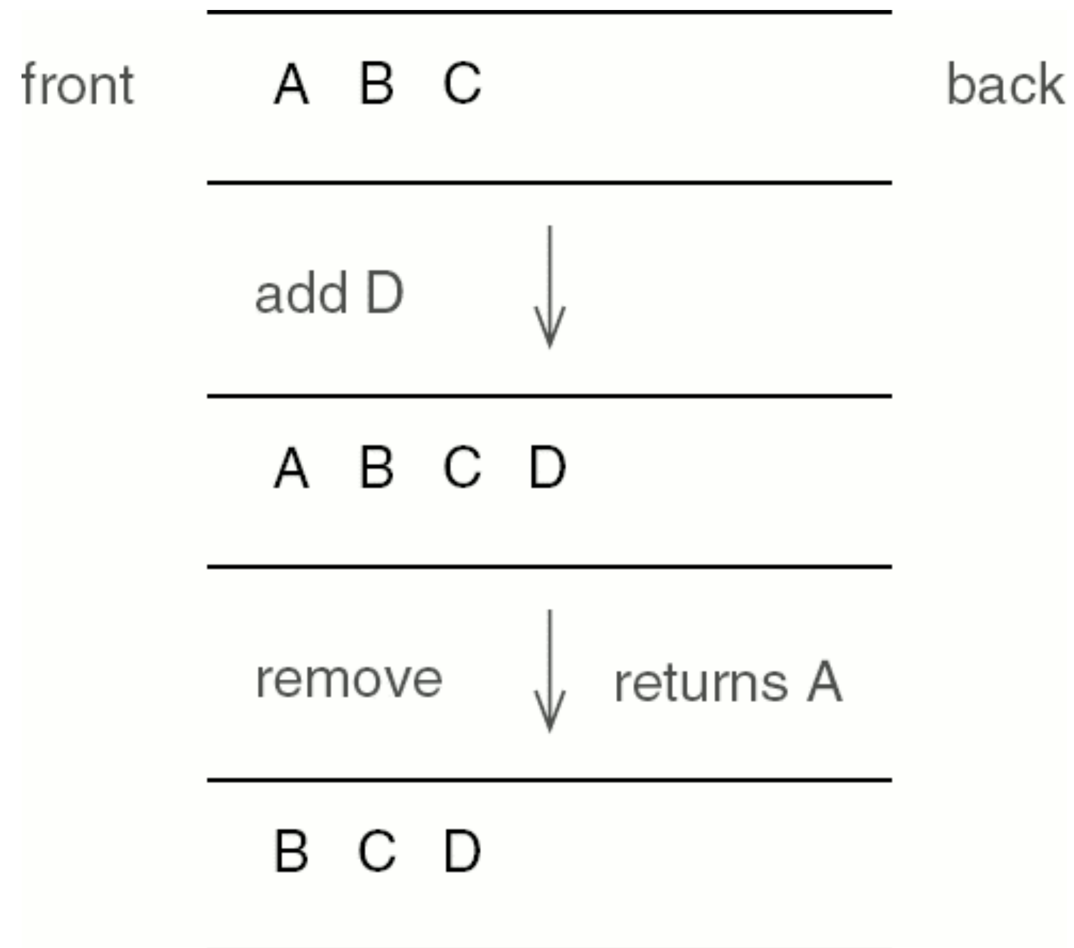
The Call Stack

```
1 Exception in thread "main"
2  java.lang.ArrayIndexOutOfBoundsException: -1
3          at Deck.deal(Deck.java:25)
4          at IdiotsDelight.deal(IdiotsDelight.java:24)
5          at IdiotsDelight.play(IdiotsDelight.java:58)
6          at IdiotsDelight.main(IdiotsDelight.java:81)
```

The Queue Interface

- *Queue*
  - Very similar to a stack.
  - Items are inserted in one end (the back) and removed from the other end (the front).
  - ***first-in, first-out***, or ***FIFO***

The Queue Interface



The Queue Interface

```
1 /** A first-in, first-out queue of Objects. */
2 public interface Queue<E> {
3
4   /** Add target to the back of this Queue. */
5   public void add(E target);
6
7   /** Return true if this Queue is empty. */
8   public boolean isEmpty();
9
10  /**
11   * Remove and return the front item from this Queue.
12   * @throws EmptyStructureException if the Queue is empty.
13   */
14  public E remove();
15
16 }
```

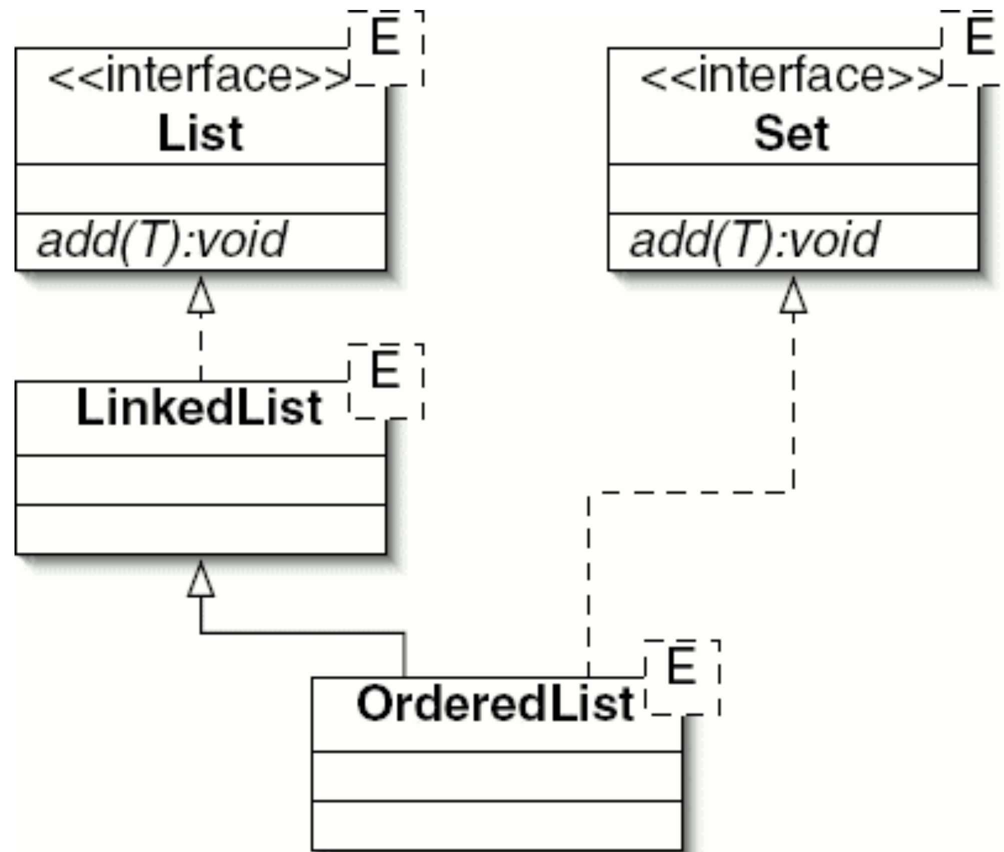
Ordered Lists

- Since a Set changes size as items are added and removed, a linked structure seems in order.
- An OrderedList is like a LinkedList, except that:
  - The elements of an *OrderedList* must implement the Comparable interfaces
  - The elements of an *OrderedList* are kept in order.
  - The *OrderedList* class implements the Set interface. It provides the methods add(), contains(), remove(), and size(). No duplicate elements are allowed.

Ordered Lists

- Extending LinkedList
  - The problem is that the LinkedList class implements the List interface, which conflicts with the Set interface.
  - The add() method from the *List* interface should add the argument target to the end of the list, even if it is already present,
    while the add() method from the *Set* interface may add target at any position, but not if it is already present.

Ordered Lists



Ordered Lists

```
1 /** A linked list of Comparable items, in increasing order. */
2 public class OrderedList<E extends Comparable<E>>
3   implements Set<E>, Predecessor<E> {
4
5   /** The first node in the list. */
6   private ListNode<E> front;
7
```

Ordered Lists

```
8   /** An OrderedList is initially empty. */
9   public OrderedList() {
10    front = null;
11  }
12
13  public ListNode<E> getNext() {
14    return front;
15  }
16
17  public void setNext(ListNode<E> next) {
18    front = next;
19  }
20
21  public int size() {
22    int tally = 0;
23    for (ListNode<E> node = front;
24         node != null;
25         node = node.getNext()) {
26      tally++;
27    }
```

Ordered Lists

```
28       return tally;
29     }
30
31     public String toString() {
32       String result = "( ";
33       for (ListNode<E> node = front;
34            node != null;
35            node = node.getNext()) {
36         result += node.getItem() + " ";
37       }
38       return result + ")";
39     }
40
41 }
```

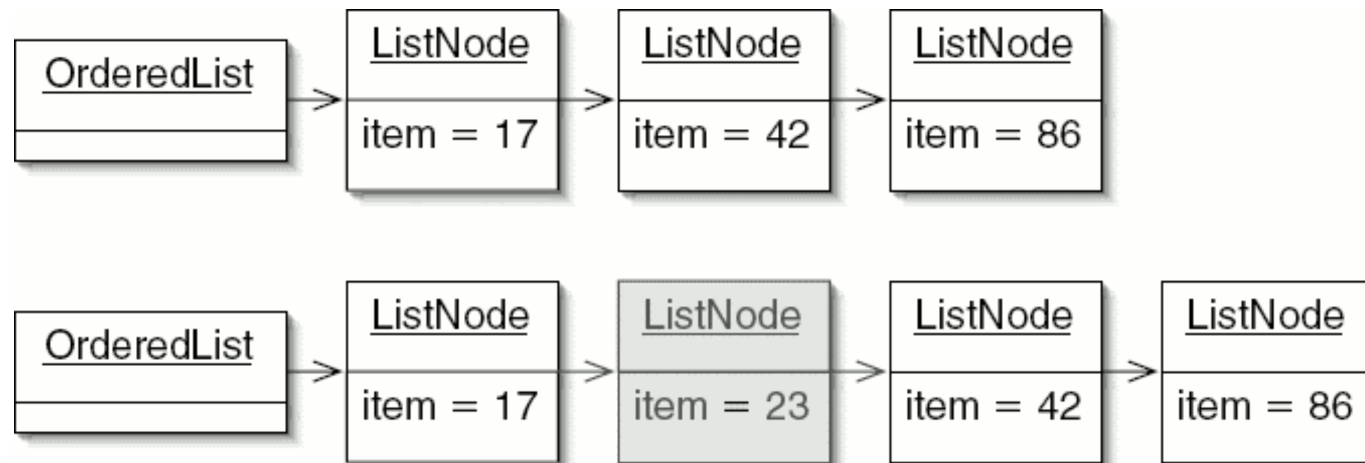
Ordered Lists

- To add something to a Set, we must first find where it belongs, then (if it is not present) put it there.
- To remove something, we must first find where it belongs, then (if it is present) remove it.
- Since the final “put it there” and “remove it” operations take constant time, all three methods have the same order of running time for a given implementation.
- The contains() method for the OrderedList class is a linear search.

Ordered Lists

```
1 public boolean contains(E target) {
2   ListNode<E> node = front;
3   while (node != null) {
4     int comparison = target.compareTo(node.getItem());
5     if (comparison < 0) {
6       return false;
7     }
8     if (comparison == 0) {
9       return true;
10    }
11    node = node.getNext();
12  }
13  return false;
14 }
```

Ordered Lists



Ordered Lists

```
1 public void add(E target) {
2   Predecessor<E> prev = this;
3   ListNode<E> node = front;
4   while (node != null) {
5     int comparison = target.compareTo(node.getItem());
6     if (comparison < 0) {
7       prev.setNext(new ListNode<E>(target, node));
8       return;
9     }
10    if (comparison == 0) {
11      return;
12    }
13    prev = node;
14    node = node.getNext();
15  }
16  prev.setNext(new ListNode<E>(target));
17 }
```

University of Zurich
Department of Informatics

Ordered Lists

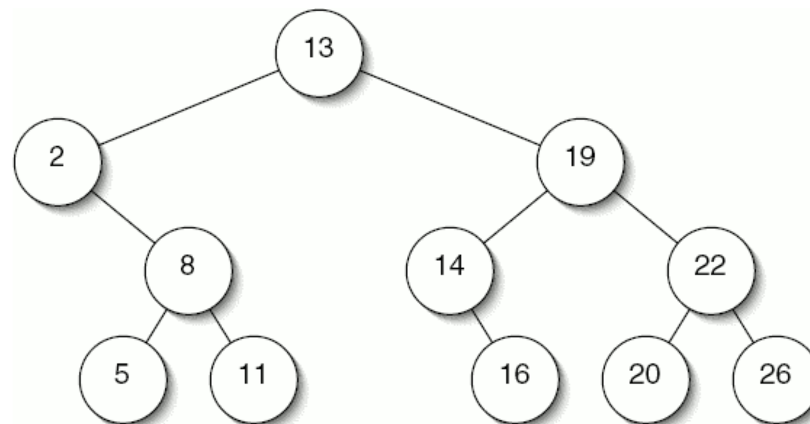
```
1 public void remove(E target) {
2   Predecessor<E> prev = this;
3   ListNode<E> node = front;
4   while (node != null) {
5     int comparison = target.compareTo(node.getItem());
6     if (comparison < 0) {
7       return;
8     }
9     if (comparison == 0) {
10      prev.setNext(node.getNext());
11      return;
12    }
13    prev = node;
14    node = node.getNext();
15  }
16 }
```

Ordered Lists

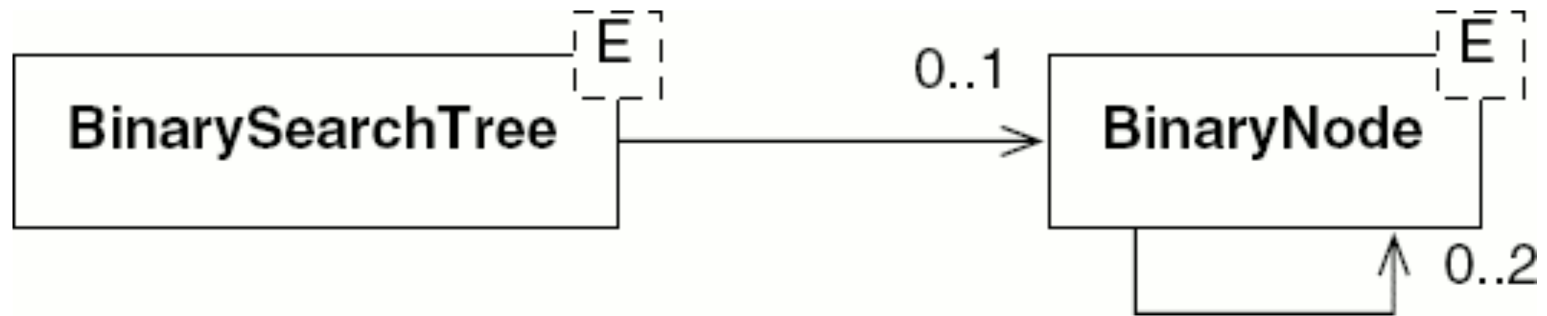
- The OrderedList data structure is easy to implement, but it requires linear time for search, insertion, and deletion.

Binary Search Trees

- *Binary search tree*
  - More efficient under some circumstances



Binary Search Trees

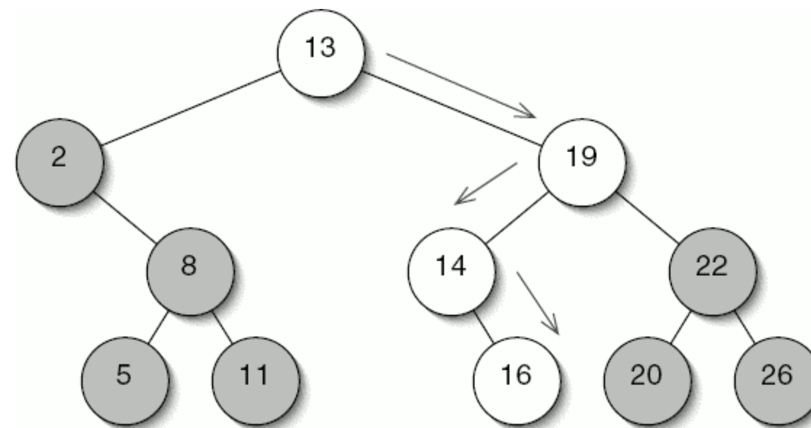


Binary Search Trees

```
1 /** A binary search tree of Comparables. */
2 public class BinarySearchTree<E extends Comparable<E>>
3   implements Set<E> {
4
5   /** Root node. */
6   private BinaryNode<E> root;
7
8   /** A BinarySearchTree is initially empty. */
9   public BinarySearchTree() {
10    root = null;
11  }
12
13  public int size() {
14    return size(root);
15  }
16
17  /** Return the size of the subtree rooted at node. */
18  protected int size(BinaryNode<E> node) {
19    if (node == null) {
20      return 0;
21    } else {
22      return 1 + size(node.getLeft()) + size(node.getRight());
23    }
24  }
25
26 }
```

Binary Search Trees

- Search



Binary Search Trees

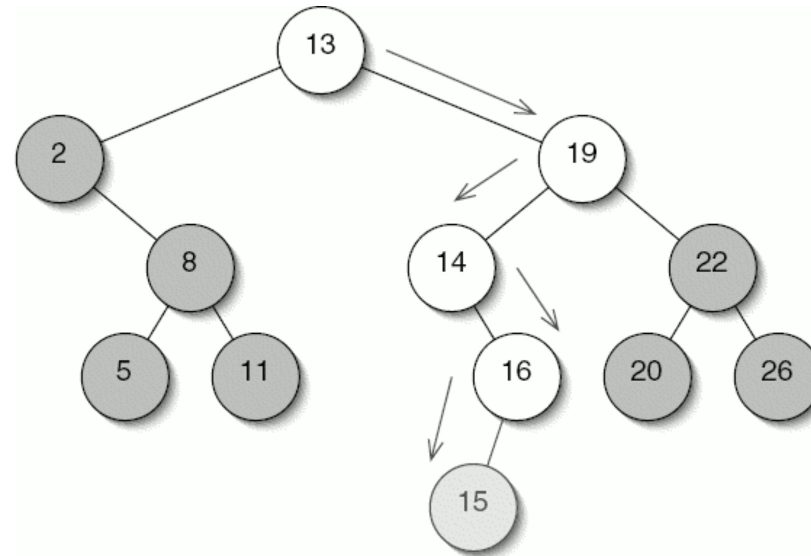
```
1 public boolean contains(E target) {
2   BinaryNode<E> node = root;
3   while (node != null) {
4     int comparison = target.compareTo(node.getItem());
5     if (comparison < 0) {      // Go left
6       node = node.getLeft();
7     } else if (comparison == 0) { // Found it
8       return true;
9     } else {                   // Go right
10      node = node.getRight();
11    }
12  }
13  return false;
14 }
```

Binary Search Trees

- Searching a binary search tree.
  - In a perfect tree, this is $\Theta(\log n)$
  - In the Anagrams program when the word file is in alphabetical order, it produces a worst case. Every new node is a right child.

Binary Search Trees

- Insertion



Binary Search Trees

- There are two complications to the code:
  - Once we reach a null node, we have forgotten how we got there. Since we need to modify either the left or right field in the parent of the new leaf, we'll need this information.
  - We need to deal with the situation in which the binary search tree is empty.

Binary Search Trees

```
 1 /**
 2  * Something which has children, such as a BinarySearchTree or a
 3  * BinaryNode.
 4  */
 5 public interface Parent<E> {
 6 
 7   /**
 8    * Return the left child if direction < 0, or the right child
 9    * otherwise.
10    */
11   public BinaryNode<E> getChild(int direction);
12 
13   /**
14    * Replace the specified child of this parent with the new child.
15    * If direction < 0, replace the left child.  Otherwise, replace
16    * the right child.
17    */
18   public void setChild(int direction, BinaryNode<E> child);
19 
20 }
```

Binary Search Trees

```
1 public BinaryNode<E> getChild(int direction) {
2   if (direction < 0) {
3     return left;
4   } else {
5     return right;
6   }
7 }
8
9 public void setChild(int direction, BinaryNode<E> child) {
10   if (direction < 0) {
11     left = child;
12   } else {
13     right = child;
14   }
15 }
```

Binary Search Trees

```
1 public BinaryNode<E> getChild(int direction) {
2   return root;
3 }
4
5 public void setChild(int direction, BinaryNode<E> child) {
6   root = child;
7 }
```

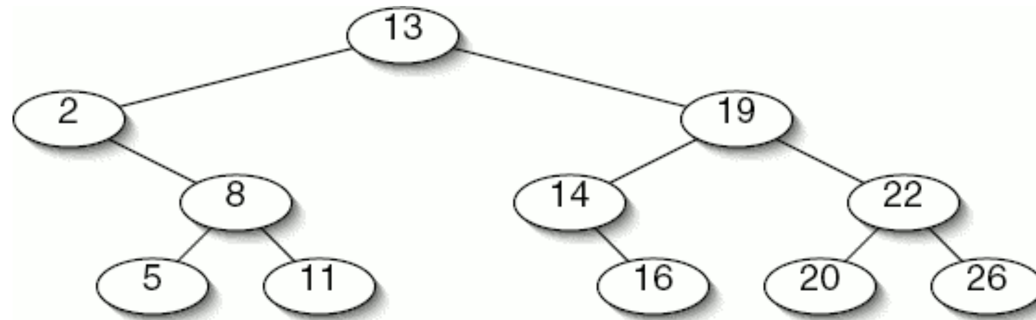
Binary Search Trees

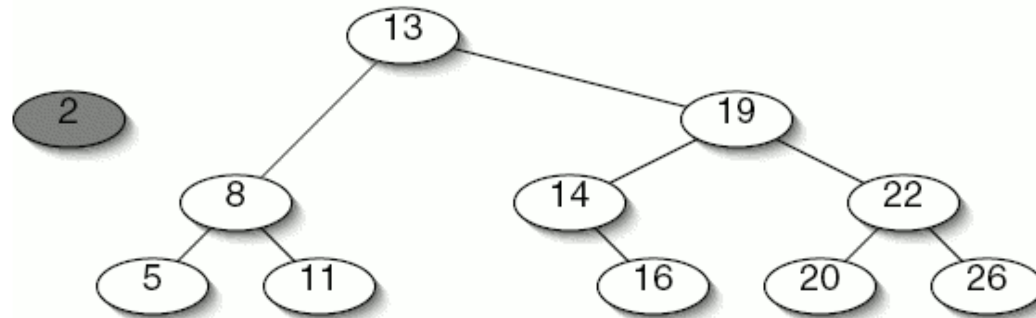
```
1 public void add(E target) {
2   Parent<E> parent = this;
3   BinaryNode<E> node = root;
4   int comparison = 0;
5   while (node != null) {
6     comparison = target.compareTo(node.getItem());
7     if (comparison < 0) {      // Go left
8       parent = node;
9       node = node.getLeft();
10    } else if (comparison == 0) { // It's already here
11      return;
12    } else {                   // Go right
13      parent = node;
14      node = node.getRight();
15    }
16  }
17  parent.setChild(comparison, new BinaryNode<E>(target));
18 }
```

Binary Search Trees

- Deletion
  - The challenge is to make sure the tree is still a binary search tree when we're done with the deletion.
  - Deleting a leaf, this is easy.
  - If the node has only one child, we just splice it out much as we would a node in a linked list.

Binary Search Trees





Binary Search Trees

- When the node we want to delete has two children.
  - We must be very careful about which node we choose to delete so that the tree will still be a binary search tree.
  - It is always safe to choose the inorder successor of the node we originally wanted to delete.
  - Find a node's inorder successor by going to the right child, then going left until we hit a node with no left child.
  - It can have a right child.

Binary Search Trees

- It is safe to replace the node we want to delete with its inorder successor.
  - It is therefore larger than anything in the left subtree and smaller than anything else in the right subtree.

Binary Search Trees

```
1 public void remove(E target) {
2   Parent<E> parent = this;
3   BinaryNode<E> node = root;
4   int direction = 0;
5   while (node != null) {
6     int comparison = target.compareTo(node.getItem());
7     if (comparison < 0) {      // Go left
8       parent = node;
9       node = node.getLeft();
10    } else if (comparison == 0) { // Found it
11      spliceOut(node, parent, direction);
12      return;
13    } else {                   // Go right
14      parent = node;
15      node = node.getRight();
16    }
17    direction = comparison;
18  }
19 }
```

Binary Search Trees

```
1  /**
2   * Remove node, which is a child of parent.  Direction is positive
3   * if node is the right child of parent, negative if it is the
4   * left child.
5   */
6  protected void spliceOut(BinaryNode<E> node,
7                           Parent<E> parent,
8                           int direction) {
9    if (node.getLeft() == null) {
10     parent.setChild(direction, node.getRight());
11   } else if (node.getRight() == null) {
12     parent.setChild(direction, node.getLeft());
13   } else {
14     node.setItem(removeLeftmost(node.getRight(), node));
15   }
16 }
```

Binary Search Trees

- We don't need special code for the case where node is a leaf, because in this situation parent.setChild(direction, **node.getRight()**);
- Is equivalent to:
  - parent.setChild(direction, **null**)

Binary Search Trees

```
1 /**
2  * Remove the leftmost descendant of node and return the
3  * item contained in the removed node.
4  */
5 protected E removeLeftmost(BinaryNode<E> node, Parent<E> parent) {
6   int direction = 1;
7   while (node.getLeft() != null) {
8     parent = node;
9     direction = -1;
10    node = node.getLeft();
11  }
12  E result = node.getItem();
13  spliceOut(node, parent, direction);
14  return result;
15 }
```

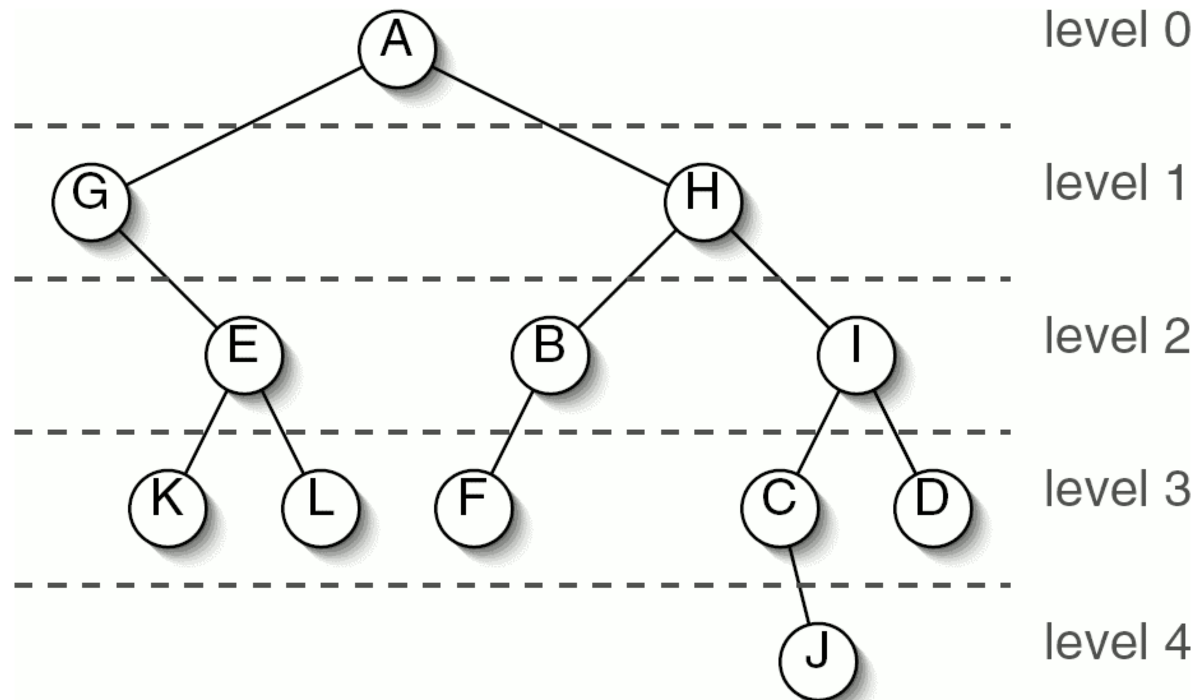
Binary Search Trees

- BinarySearchTrees should not be used in the plain form explained here.
  - The worst-case, running time is linear
  - Worst case are not uncommon.

Tree Traversal

- Four meaningful orders in which to traverse a binary tree.
  - *Preorder*
  - *Inorder*
  - *Postorder*
  - *Level order*

Tree Traversal



Traversal Order	Order in which Nodes are Visited
Preorder	AGEKLHBFICJD
Inorder	GKELAFBHCJID
Postorder	KLEGFBJCDIHA
Level order	AGHEBIKLFCDJ

Tree Traversal

```
1 /**
2  * Return a String representation of the tree rooted at this node,
3  * traversed preorder.
4  */
5 public String toStringPreorder() {
6   String result = "";
7   result += item;
8   if (left != null) {
9     result += left.toStringPreorder();
10  }
11  if (right != null) {
12    result += right.toStringPreorder();
13  }
14  return result;
15 }
```

Tree Traversal

```
1 /**
2  * Return a String representation of the tree rooted at this node,
3  * traversed inorder.
4  */
5 public String toStringInorder() {
6   String result = "";
7   if (left != null) {
8     result += left.toStringInorder();
9   }
10  result += item;
11  if (right != null) {
12    result += right.toStringInorder();
13  }
14  return result;
15 }
```

Tree Traversal

```
1 /**
2  * Return a String representation of the tree rooted at this node,
3  * traversed postorder.
4  */
5 public String toStringPostorder() {
6   String result = "";
7   if (left != null) {
8     result += left.toStringPostorder();
9   }
10  if (right != null) {
11    result += right.toStringPostorder();
12  }
13  result += item;
14  return result;
15 }
```

Tree Traversal

```
1 /**
2  * Return a String representation of the tree rooted at this node,
3  * traversed preorder.
4  */
5 public String toStringPreorder() {
6   String result = "";
7   Stack<BinaryNode<E>> stack = new ArrayStack<BinaryNode<E>>();
8   stack.push(this);
9   while (!(stack.isEmpty())) {
10    BinaryNode<E> node = stack.pop();
11    result += node.item;
12    if (node.right != null) {
13      stack.push(node.right);
14    }
15    if (node.left != null) {
16      stack.push(node.left);
17    }
18  }
19  return result;
20 }
```

University of Zurich
Department of Informatics

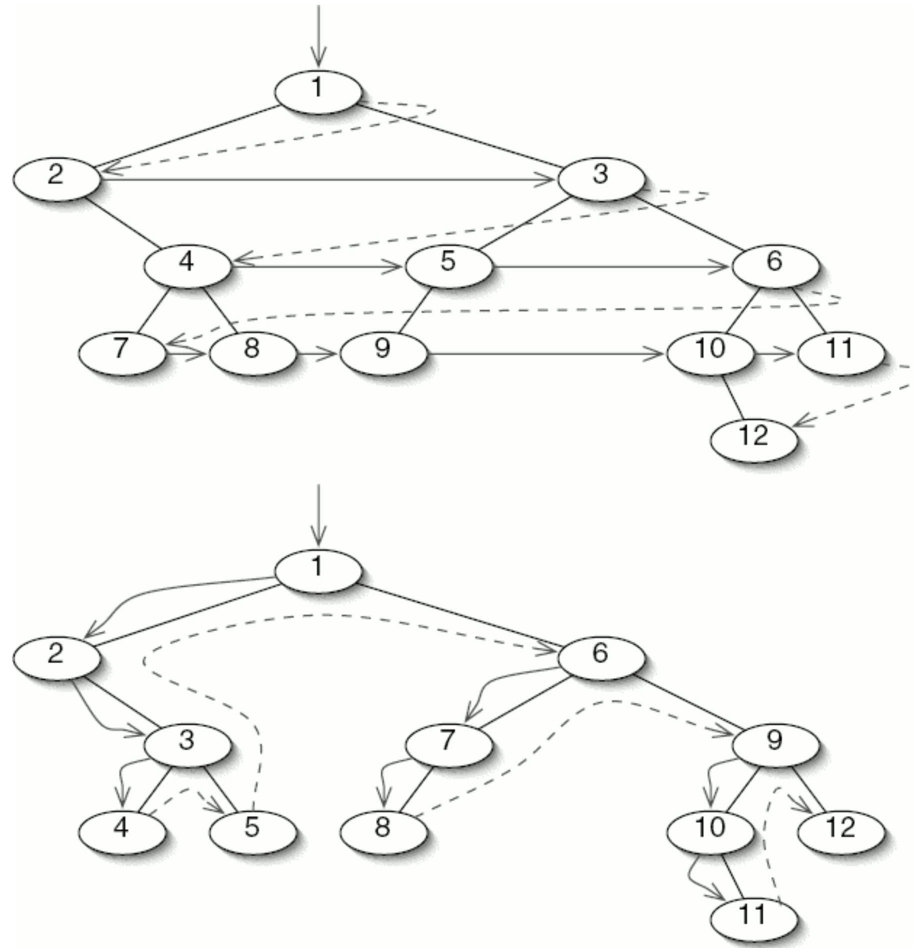
Tree Traversal

```
1 /**
2  * Return a String representation of the tree rooted at this node,
3  * traversed level order.
4  */
5 public String toStringLevelOrder() {
6   String result = "";
7   Queue<BinaryNode<E>> q = new ArrayQueue<BinaryNode<E>>();
8   q.add(this);
9   while (!(q.isEmpty())) {
10    BinaryNode<E> node = q.remove();
11    result += node.item;
12    if (node.left != null) {
13      q.add(node.left);
14    }
15    if (node.right != null) {
16      q.add(node.right);
17    }
18  }
19  return result;
20 }
```

Tree Traversal

- Level order traversal is sometimes called *breadth-first*.
- The other traversals are called *depth-first*.
- Traversal takes $\Theta(n)$ in both breadth-first and depth-first.
- Memory usage in a perfect tree is $\Theta(\log n)$ in depth-first and $\Theta(n)$ in breadth-first traversal.

Tree Traversal



General Trees

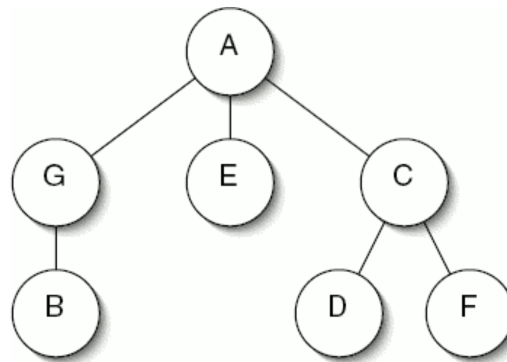
- General Tree
  - General trees differ from binary trees in three ways:
    - A node in a general tree may have more than two children.
    - General trees cannot be empty. This restriction is made to avoid having to distinguish between a node with no subtrees and a node with several empty subtrees, which would be drawn identically.

General Trees

- A node in a general tree has a (possibly empty) sequence of children, rather than a certain number of “slots” to fill.
- Binary trees
  - a tree with a left subtree but no right subtree
  - a tree with a right subtree but no left subtree.
- No such distinction is made for general trees

General Trees

- Inheritance diagrams showing the relationships between classes is a general tree.



General Trees

- Simplest is to represent each node as an item.
  - ***Array of children*** or ***list of children***.
  - ***First-child***, ***next-sibling***
    - A less intuitive but more space-efficient representation has each node keeping track of its first child and its next sibling.

General Trees